Playing video from a CFFA3000

Kris Kennaway
KansasFest 2021

Overview

- Full-screen, double hi-res video, with audio, on a 1MHz Apple //e
- using a CFFA3000 compact flash adapter

Builds on previous projects

- Video streaming via ethernet (][-vision, KFest 2019)
- Improvements in DHGR image conversion (][-pix, KFest 2021)

Describe how I did it

- warning: lots of technical detail!
- sample videos - see #video-cffa3000 on Discord, or links at end

Past work

Lots of other related work by others

- playing videos in lo-res
- particular black & white videos in hi-res from CFFA3k (“Bad Apple”)
- ...or 5.25” floppy (!)

This approach allows:

- much higher video bandwidth
  - full-colour, double hi-res video
- speaker audio playback (5.4-bit PWM @ 20KHz)
- fully general video pipeline for transcoding from modern machine to play on Apple II using CFFA3k (or Uthernet II)

How does the CFFA3000 *really* work?

- implements standard device driver APIs (e.g. SmartPort) for block-level I/O
  - and emulates a Disk II
- ...otherwise undocumented
- guess: slot firmware must be using a lower level mechanism to talk to the hardware
- are there opportunities to make use of this for increased I/O performance?

Reverse engineering the CFFA3k slot firmware

- dumped $Cn00 slot firmware, and $C800..$CFFF extended firmware
- reverse engineered code starting from standard SmartPort I/O entry points
- traced the main I/O command loop
  - No use of $C0xx I/O soft switches
  - All I/O is via memory addresses in the extended firmware address space ($CFxx)
  - “ROM” $Cnxx memory is actually RAM-backed!
  - Can modify the “firmware” (as seen by Apple II) dynamically at runtime
    - firmware itself makes heavy use of this

Core I/O processing loop

- synchronization protocol for coordinating/communicating with onboard HW
  - shared memory semaphore; Apple II and onboard HW share memory
- dispatch loop:
  - issue SmartPort I/O command to hardware
  - HW drives 6502 through sequence of operations to complete processing of command
    - some of them involve HW dynamically modifying the firmware address space to map in code, then telling 6502 to jump to it
    - hard to get a complete firmware dump, but core logic is always mapped
- modified this dispatch loop to record a trace of operations
  - not timing critical, insert a JMP to my own code elsewhere in memory

Block I/O reads

- read operations
  - copy from $c800.$c9ff into caller’s requested buffer
  - clean up and return
- this means that $c800.$c9ff in firmware space is used as an I/O buffer!
- when 6502 issues a block read request, after some time the contents magically appear at $c800.$c9ff
- copied from there to caller’s I/O buffer
- copying to main memory is slow
  - fastest possible fully unrolled loop is 4096 cycles
    - LDA $c800
    - STA $2000
    - LDA $c801
    - STA $2001
    - … ; 512 times

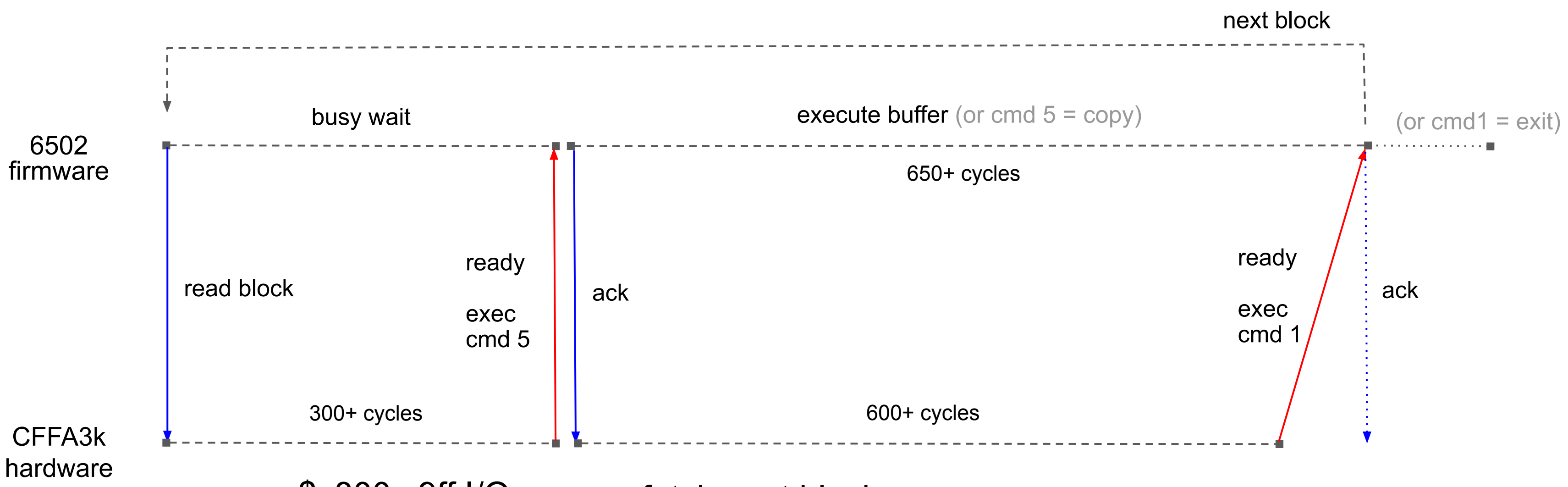
How can I make use of this internal buffer?

- could write 6502 code to access the I/O buffer and take some action
  - e.g. write a pixel to screen memory
- but anything we can do would be slower than just copying the entire buffer into screen memory
- 6502 can only read a byte from memory once per 4 cycles
- 122800 cycles for unrolled LDA/STA loop to store double hi-res frame as fast as possible
  - 8.3 frames/second
  - ...assuming I/O is infinitely fast
  - 92KB of code just for unrolled screen copies

Is this the best we can do?

- Can’t read from the buffer faster than this under 6502 program control
- ...but 6502 itself can access the buffer faster -- by executing it
- load 512 bytes of code into the buffer, execute it to … do stuff
  - LDA #$FF ; we know what value to store, don’t have to load from memory
  - STA $2000
  - STA $2001 ; can store a value in multiple locations
  - STA $3f02 ; ...which don’t have to be contiguous in memory
  - …
- 2+4+4+4=14 cycles to store 3 bytes, instead of 24
  - ~2x faster
- but: 2+3+3+3=11 bytes to represent 3 bytes of screen contents, instead of 3
  - ~3x less space efficient
- 512 bytes of such 6502 code executes in ~650 cycles
- **Is this better?**

CFFA read sequence



next block

busy wait

execute buffer (or cmd 5 = copy)

(or cmd1 = exit)

6502
firmware

650+ cycles

read block

ready

exec
cmd 5

ack

ready

exec
cmd 1

ack

300+ cycles

600+ cycles

CFFA3k
hardware

unmap $c800.c9ff I/O buffer

map block from cache, or load from disk

prefetch next block

Is this better? Yes!

- Note that the minimum I/O prefetch time (~600 cycles) is almost exactly how long it would take to execute code in the buffer (~650 cycles)!
- So by the time we finish executing, the CFFA will have (usually) finished prefetching the next block
  - we only need to wait 300 cycles for it to be mapped
  - we can't do much about this, at least with current firmware
- we can execute up to 2TB of 6502 code, paged in 512 byte chunks, at ⅔ of native CPU speed
  - ~650 cycles every 950
- Reads data at about **533 KB/sec**
  - cf 78KB/sec using SmartPort API; 6.8x faster

Playing video

- Up to 128 screen updates/page ~ 145000/sec
- ~ 9.4 full double hi-res screen updates/sec
    - cf <8/sec for the “full frame update” approach
- i.e. a bit better in the worst case
- *much* better in typical case
    - most videos don’t change every pixel every frame
    - we can change as many or as few pixels as we like
- we can do other things as well
    - ...like toggle the speaker?
    - requires exact cycle counting

Strategy

- unroll the video into straight-line 6502 code that updates screen memory and flips display switches
- ...while toggling the speaker at exact cycle timings to produce audio
- package into 512-byte chunks, stitched together with I/O code
- 533KB/sec of data → 31MB of code per minute of playback
  - ...but since we're using a CF with GB's of storage, this is not a problem

Strategy

- unroll the video into straight-line 6502 code that updates screen memory and flips display switches
- ...while toggling the speaker at exact cycle timings to produce audio
- package into 512-byte chunks, stitched together with I/O code
- 533KB/sec of data → 31MB of code per minute of playback
    - ...but since we’re using a CF with GB’s of storage, this is not a problem

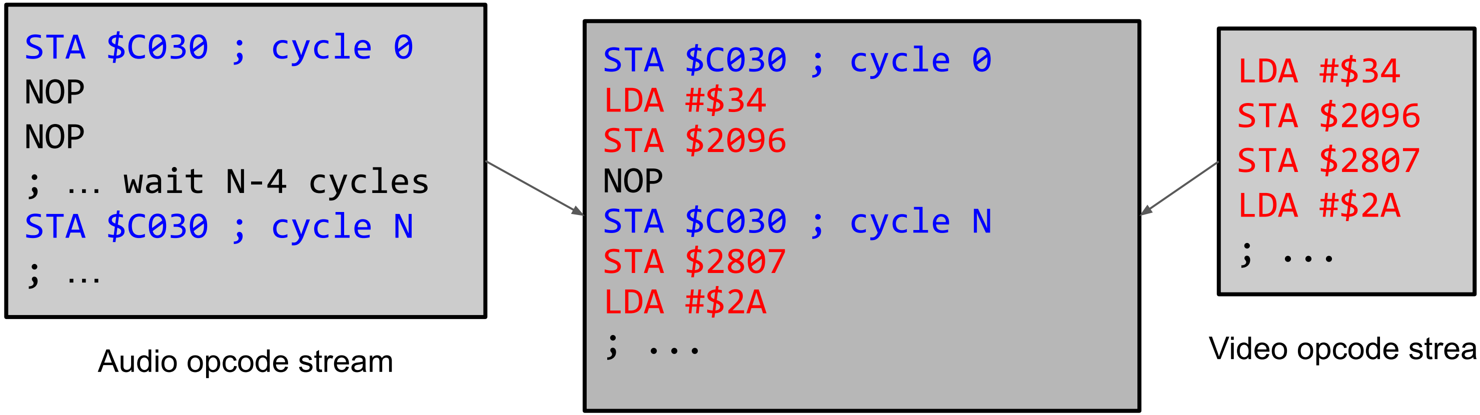
...We just need to write the world’s largest 6502 program.

That sounds hard, let’s write a program to do it

- Actually I already wrote (most of) this program in 2019
-][-Vision: Uthernet II video player
    - multiplexes video and audio stream into a native Apple II format
    - handles PWM audio encoding
    - understands Apple II graphics memory structure
    - encodes image frames as (D)HGR images
    - delta encoding and prioritization of changes between image frames
        - sends the bytes that will make largest visual difference to image first
        - in case we run out of time to send them all
- Needed to swap output representation to use generated 6502 code
    - instead of a bytecode representation of the video stream
- Also swapped out DHGR image encoding to use][-pix

Tricky parts (1)

- PWM audio requires toggling speaker at precise cycle intervals
- need to interleave STA $C030 with instruction stream that performs video updates/housekeeping
  - while maintaining exact cycle timings
- wrote code to do this opcode stream interleaving



```
STA $C030 ; cycle 0
NOP
NOP
; … wait N-4 cycles
STA $C030 ; cycle N
; …
```

Audio opcode stream

```
STA $C030 ; cycle 0
LDA #$34
STA $2096
NOP
STA $C030 ; cycle N
STA $2807
LDA #$2A
; ...
```

Merged opcode stream

```
LDA #$34
STA $2096
STA $2807
LDA #$2A
; ...
```

Video opcode stream

Bonus - audio quality

- normally for PWM audio playback from memory you can’t fetch samples quickly enough to play back at 22Khz
  - Fetch an 11KHz sample and play it twice
- Here we don’t have to fetch samples because we unroll the code at generation time
- Can play audio at true 20KHz
  - ...mostly

Tricky parts (2)

- We need to keep playing audio during the 300+ cycles of I/O dead time
    - while driving the CFFA hardware and waiting for it to map the next block
    - ...and our code buffer is unmapped during this time!
- While we’re executing our 512 byte block, queue up audio samples that we can fetch and play during the CFFA I/O (idea: Lucas Scharenbroich)
    - push audio sample values onto stack
    - carve up CFFA I/O code into ~100-cycle segments
    - generate N variants of these code segments
        - each toggles the speaker at (N, 50-N) cycle intervals
        - while driving CFFA I/O, fetching the next sample and chaining to next I/O segment
    - reuse the same code interleaving technique to generate these variants
    - these samples are played back twice at 20KHz like usual for in-memory playback
- Has more overhead: ~450 cycles instead of 300
- ...but we get audio with our video!

Lessons learned

- look at the physical hardware, don’t just dive into software
- reverse engineered low-level I/O protocol from first principles
    - a lot of it was in the datasheets :-/
- Modern Apple II peripherals tend to make use of off-the-shelf components that perform a lot of the heavy lifting
    - cf fully custom logic
    - these are usually well documented, and often exposed directly to Apple II access
- Timing measurement trick:
    - to measure speed of timing loops, insert a STA $C030
    - then measure audio frequency with smartphone app
    - also lets you hear if you have cycle non-exactness

CFFA bugs

- SmartPort reads > 32MB aren’t handled correctly
  - ProDOS only supports 32MB volumes
  - but SmartPort should support 2^24 blocks = 8GB
  - workaround (Dave Lyons): use Extended SmartPort commands - not supported by //e firmware, but supported internally and used by //gs firmware
- Writes to certain $CFxx memory locations causing nearby **reads** to become corrupted
  - won’t affect normal operation of the card
  - timing/electrical issue in the HW?
  - problem with my particular board?

What’s next?

- finish cleaning up code and merge back to][-Vision
- Support running on //gs -- different firmware, should just need minor changes
- See what can be done with other mass storage devices
- Optimizations and algorithmic improvements to video encoding
- Other applications for paged code technique?

Links:

- (in future) code: https://github.com/KrisKennaway/ii-vision
- download video files
    - https://www.dropbox.com/sh/nzh7iv6h97g3zbc/AADMDfXMIN5tdvexM1RpIJ1ha?dl=0

Bonus: Booti

- uses a CH376 USB controller on a daughterboard
- [+] hardware supports reading 65KB at a time, not just 512b
- [+] data is streamed via $C0xx I/O port (like Uthernet II)
- [-] I/O is fully synchronous; no hardware prefetch
    - would make audio difficult, although maybe the queueing technique could help
- [?] haven’t measured read throughput yet
    - but for CFFA3k, USB access is *much* slower than CF

Bonus: underlying HW seems to expose much more general capabilities

- full R/W access to the USB filesystem, not just disk image file
- more general USB device I/O?